departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concurrency and Parallelism
(Concorrência e Paralelismo – CP 11158)

Lecture 12
— Performance Analytics —

Some Parallel Patterns

- Source: Williams, A (2011) "Picking Patterns for Parallel Programs (Part 1)", Overload, 105, 15-17.

- Loop Parallelism
- Fork/Join
- Pipelines
- Actor
- Speculative Execution

Loop Parallelism

- Problem
  - There is a ***for*** loop that operates on many independent data items
- Solution
  - Parallelize the *for* loop
  - The operation should depend only on the loop counter, and the individual loop iterations should not interact
- Positives
  - Scales very nicely
  - Very common
- Negatives
  - Overhead of setting up the thread
  - Avoid if there is interaction as the individual iterations may execute in any order

Fork/Join

- Problem
  - The task can be broken into two or more parts that can be run in parallel
- Solution
  - Use a thread for each part
  - This can also be recursive
- Positives
  - Handles part interaction better than Loop Parallelism
  - Works best at the top level of the application
- Negatives
  - Needs to be managed centrally so that hardware parallelism is utilized efficiently
  - Overhead of threads
  - Bursty parallelism
  - Uneven workloads

Pipelines

- Problem
  - There is a set of tasks to be applied in turn to data, FIFO order
    - This problem shows up in sensor data processing a lot
- Solution
  - Set up the tasks to run in parallel
  - Fill the input queue
- Positives
  - Adapted well to heterogeneous hardware configurations
- Negatives
  - Setting it up
  - Ensuring that the tasks have similar durations to avoid a rate-limiting step
  - Cache interaction during transfers between pipeline stages

Actor

- Problem
  - Message-passing object-orientation with concurrency
  - Message sending is asynchronous
  - Response processing uses call-backs
- Solution
  - Objects communicating (only) via message queues
- Positives
  - Actors can be analyzed independently
  - Avoids data races
- Negatives
  - Setup and queue management overhead
  - Not good for short-lived threads
  - Not an ideal communications mechanism
  - Limited scalability

Speculative Execution

- Problem
  - There's an optional path that may be required for a solution, but it takes a lot of time
- Solution
  - Start it early and cancel it if it's not needed
  - This is part of how the human brain works
- Positives
  - Exploits parallelism
  - Likely to improve performance
- Negatives
  - Wastes energy and resources
  - Interferes with other use of parallelism

Performance analytics

- The two main reasons for implementing a parallel program are to obtain better performance and to solve larger problems
- Performance can be both modeled and measured
- Illustrate some of the factors that influence the performance of a parallel program

Sequential Computing Time

- Consider a computation consisting of three parts: **a setup section, a computation section, and a finalization section**
- The total running time of this program on one processor is then given as the sum of the times for the three parts

$$T_{total}(1) = T_{setup} + T_{compute} + T_{finalization}$$

Parallel Computing Time

- Assume that…
  - The setup and finalization sections cannot be carried out concurrently with any other activities
  - The computation section can be divided into tasks that will run independently on as many processors as are available
    - with the same total number of computation steps as in the original computation
- The time for the full computation on P processors can therefore be given by

$$T_{total}(P) = T_{setup} + \frac{T_{compute}(1)}{P} + T_{finalization}$$

  - The idea that computations have a serial part (for which additional PEs are useless) and a parallelizable part (for which more PEs decrease the running time) is realistic

Speedup

- An important measure of how much additional PEs help is the relative speedup S
  - Describes how much faster a problem runs in a way that normalizes away the actual running time

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)}$$

Efficiency

- The efficiency E is the speedup normalized by the number of PEs

$$E(P) = \frac{S(P)}{P} = \frac{T_{total}(1)}{P \times T_{total}(P)}$$

Serial Fraction

- Ideally, speedup to be equal to P, the number of PEs
  - This is sometimes called perfect linear speedup
    - This can rarely be achieved because times for setup and finalization are not improved by adding more PEs, limiting the speedup
    - The terms that cannot be run concurrently are called the serial terms

- The running times of serial terms is the *serial fraction*, denoted γ

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)}$$

Parallel Fraction

- The fraction of time spent in the parallelizable part of the program is then (1 — γ)
- We can thus rewrite the expression for total computation time with P PEs as

$$T_{total}(P) = \underbrace{\gamma \times T_{total}(1)}_{T_{setup} + T_{finalization}} + \frac{\overbrace{(1-\gamma) \times T_{total}(1)}^{T_{compute}(1)}}{P}$$

Amdahl's Law

- Now, rewriting S in terms of the new expression for $T_{total}(P)$, we obtain the famous Amdahl's law

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)} = \frac{T_{total}(1)}{(\gamma + \frac{1-\gamma}{P}) \times T_{total}(1)} = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

Maximum Speedup (infinite PEs)

- An ideal parallel algorithm, with no overhead in the parallel part, the speedup should follow the equation from Amdahl's Law
- What happens to the speedup if we take our ideal parallel algorithm and use a very large number of processors?
- Taking the limit as P goes to infinity in our expression for S yield

$$S = \frac{1}{\gamma}$$

The End